ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/329107805

Flipper 2.0: A Pragmatic Dialogue Engine for Embodied Conversational Agents

Conference Paper · November 2018

DOI: 10.1145/3267851.3267882

CITATIONS
27

READS
698

7 authors, including:

Jelte van Waterschoot
Saxion
7 PUBLICATIONS **79** CITATIONS
SEE PROFILE

Merijn Bruijnes
Utrecht University
41 PUBLICATIONS **431** CITATIONS
SEE PROFILE

Dennis Reidsma
University of Twente
180 PUBLICATIONS **3,821** CITATIONS
SEE PROFILE

Daniel Davison
University of Twente
23 PUBLICATIONS **342** CITATIONS
SEE PROFILE

All content following this page was uploaded by Mariet Theune on 27 November 2018.

The user has requested enhancement of the downloaded file.

Flipper 2.0

A Pragmatic Dialogue Engine for Embodied Conversational Agents

Jelte van Waterschoot
j.b.vanwaterschoot@utwente.nl
University of Twente
Enschede, Netherlands

Merijn Bruijnes
m.bruijnes@utwente.nl
University of Twente
Enschede, Netherlands

Jan Flokstra
jan.flokstra@utwente.nl
University of Twente
Enschede, Netherlands

Dennis Reidsma
d.reidsma@utwente.nl
University of Twente
Enschede, Netherlands

Daniel Davison
d.p.davison@utwente.nl
University of Twente
Enschede, Netherlands

Mariët Theune
m.theune@utwente.nl
University of Twente
Enschede, Netherlands

Dirk Heylen
d.k.j.heylen@utwente.nl
University of Twente
Enschede, Netherlands

ABSTRACT

We present a new dialogue engine called Flipper 2.0 (Flipper) which aims to help developers of embodied conversational agents (ECAs) to quickly and flexibly create dialogues. Flipper provides a technically stable and robust dialogue management system to integrate with other components of ECAs such as behaviour realisers. We compare Flipper with state-of-the-art dialogue design systems. We describe the details of our dialogue engine, how it handles dialogue management and how it supports the authoring of dialogues. We demonstrate the use of the dialogue engine with examples of design patterns and discuss practical applications. Finally we give recommendations on the cases in which it is beneficial to use Flipper.

CCS CONCEPTS

• **Human-centered computing** → **Natural language interfaces**; **Systems and tools for interaction design**; *User interface toolkits*; • **Information systems** → *Open source software*;

KEYWORDS

dialogue manager, dialogue engine, dialogue design, pragmatics, embodied conversational agent

ACM Reference Format:
Jelte van Waterschoot, Merijn Bruijnes, Jan Flokstra, Dennis Reidsma, Daniel Davison, Mariët Theune, and Dirk Heylen. 2018. Flipper 2.0: A Pragmatic Dialogue Engine for Embodied Conversational Agents. In *Proceedings of the 18th Annual Conference on Intelligent Virtual Agents.* ACM, Sydney, NSW, Australia, 8 pages. https://doi.org/10.1145/3267851.3267882

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
IVA '18, November 2018, Sydney, Australia
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6013-5/18/11.
https://doi.org/10.1145/3267851.3267882

1 INTRODUCTION

The task of building multi-modal dialogue systems for embodied conversational agents (ECAs) in large multi-partner research projects is not trivial. Such systems need to handle complex, emergent, multi-modal dialogues, be continuously responsive, and deal with unpredictable user input. The reality is that in such projects the dialogue system consists of, and interfaces with, several specialised components from different partners, each with their own technical framework. The ideal dialogue system has two dimensions: it needs to (1) support researchers to achieve the complexity of the emerging dialogues that current projects strive for; and (2) support the quick creation of (partially) functional prototypes that can demonstrate and/or evaluate the effect of design choices or of prospective technical components on the ECA early in the project.

The first version of Flipper was used for performing dialogue management in the SEMAINE[1] project [22]. We have upgraded Flipper to navigate the abovementioned two dimensions and present Flipper 2.0[2], a declarative language and interpreter specifically designed to quickly and iteratively create a dialogue manager for an ECA. Towards that goal we have designed Flipper with the following capabilities. (1) With Flipper, basic dialogues can be created with minimal overhead. (2) Flipper can switch between (a) delegating a task to an external specialised component (e.g. sensor interpretation or decision making); and (b) simulating prospective external components from within the dialogue templates as a temporary placeholder until the component exists. (3) Flipper supports choosing along the spectrum between (a) robust, scalable and well-defined declarative models of dialogues; and (b) pragmatic 'hacking-stuff-together' and 'wizarding' to try out the effects of certain dialogue paradigms before actually modelling them properly. This can help early in the project to show how a dialogue with the ECA will emerge. It also helps to make decisions that are informed by the reality of the distributed ECA technology that is available or that will be developed. Early demonstrations demand a pragmatic

[1] https://semaine-db.eu/
[2] From now on, Flipper 2.0 is denoted as Flipper

approach while at some point the pragmatic developments and the lessons learned need to be consolidated into an ECA system that is robust and scalable. (4) Information in Flipper can be stored in a persistent database which enables, for instance, a robust consistency between interactions over time. (5) Flipper can process information from input sensors in parallel, handle decision making, and create and send output, making the ECA continuously responsive in a dialogue. (6) Flipper can communicate with external components; it currently supports seven middleware communication platforms and it is easy to add other methods of communication. (7) Finally, over the course of several national and international research projects we have created a set of 'design patterns'. In these design patterns we show how we solve in a robust and scalable way the typical situations and technical problems that occur when creating a dialogue system. We will make these available together with the software and highlight some in this paper.

In Section 2 we explain our view on dialogue systems and discuss related work on dialogue management and designing dialogues. In Section 3 we discuss the technical details of Flipper. In Section 4 we show some examples and design patterns of using the dialogue engine. In Section 5 we point at some work that has been done with Flipper in ongoing and earlier research projects. Finally, we discuss the current limitations and future development of the dialogue engine in Section 6 and present our conclusions in Section 7.

2 BACKGROUND

ECAs consist of multiple technical components that can be roughly divided into three 'pillars of tasks': sense, think, and act. In an interaction, 'sensing' components are tasked with processing and interpreting the human's language and social signal behaviour (e.g. the user's mouth corners move up, meaning a smile). This information is used by the agent to 'think' about the behaviour of the user (e.g. the user liked the joke I just told) in order to decide what is an appropriate response in the context (e.g. laugh with the user to create rapport). This response behaviour is displayed ('acted') by the body of the ECA. Each component in each pillar has a distinct task that it performs in order for an ECA to function in a social interaction.

We distinguish within the 'thinking' component of an ECA a division of three parts: a dialogue engine, a dialogue manager and dialogues. The **dialogue manager** is the part of an ECA that deals with how the agent behaves in an interaction. It is a collection of rules that control the flow and state of the conversation [12]. It does so in response to the input of the user, and the goals and beliefs of the ECA. The **dialogue engine** is the machinery with which it is possible to create a dialogue manager. This can be done in a regular programming language, or in a system that interprets declarative dialogue specifications to control a dialogue, or a mixture of the two. Authors of **dialogues** are then required to write content (a dialogue structure within the domain that the agent knows about and can converse about and that contains all the behaviours that the agent can decide to do) and add this to the dialogue system. Together, the dialogue engine, dialogue manager and dialogues make up the complete dialogue system.

Choosing a tool to develop dialogues for your ECA has a great impact on the type of interaction. An overview of different tools that are currently available is provided in Table 1 . We review seven aspects of each dialogue design tool: information processing, the interface to author dialogues, the support for linking an embodied agent, the design paradigm, how dialogue control is organised, the support for different types of interaction management and the inclusion of design patterns for authoring dialogues.

Information processing is the way in which the context of the dialogue is stored and processed. In most tools this is captured in either states or a network. A state-based approach is easier to interpret and for authoring dialogues, than a network-based approach. However, if you have much data available, a network would be very convenient to capture all relevant information without explicitly stating what is relevant. Flipper uses information-state update rules, similar to the approaches in TrindiKit [12] and the VHToolkit [10]. The information state update approach is useful for keeping control of the dialogue flow without declaring all possible dialogue states. Commercial cloud-based services such as LUIS.AI, Wit.ai, DialogFlow, Watson, Lex and Recast.ai all use a neural network for processing the information[3], which is a useful approach for learning from large datasets containing text or conversations [2, 4].

Interface of Authoring is the process of designers creating interactions for their ECA. The accessibility of authoring is important for designers to use your tool. The VHToolkit, with the NPCEditor [14], Visual SceneMaker [7] and HALEF [17], with OpenVoiceXML, provide a graphical user interface for editing the dialogue. Other tools, like Flipper, IrisTK [21] and OpenDial [15] use a declarative way of defining the dialogue in XML. In the commercial cloud-services, designers can use a web-interface to author dialogues, where they provide user input and an appropriate agent response, marking the intents and entities in the utterances. In the VHToolkit [10], WAMI [9] and Flipper scripting is also possible for less restricted authoring.

Embodiment is an ECA's capability to perform both verbal and non-verbal behaviours. Most of the tools support this and it is a necessity for developing an ECA. IrisTK [21], Visual SceneMaker [7] and the VHToolkit [10] come packed with an embodiment. Others, like RavenClaw [1], Disco [19], OpenDial [15] and Flipper have interfaces available for embodiment. The commercial tools are harder to link to an embodiment, due to the restricted intent-entity mapping. HALEF [17] is less suitable for embodiment, due to its focus on telephone-conversation.

Developing dialogues can be done via a bottom-up pragmatic approach, a more theory-driven robust manner or a mixed approach, which are the **design paradigms**. PyDial [24] and the commercial tools like DialogFlow are more on the pragmatic side of the design paradigm scale, for quickly developing content with conversational data. Tools like RavenClaw [1] and Disco [19] require a theory-driven approach due to their hierarchical way of processing information and have only limited conversational data available. The emphasis of Flipper is on using a pragmatic approach when starting to develop dialogues, though for more complex dialogues theory-driven development is also possible, similar to the design paradigm in OpenDial[15].

The **dialogue control** can be either single or distributed (multi-agent) [6]. In IrisTK [21] a single component is responsible for the

[3] luis.ai, wit.ai, dialogflow.com, ibm.com/watson/, aws.amazon.com/lex/ and recast.ai

Table 1: An overview of different dialogue design tools. For each tool is listed which architecture is used for information processing, the authoring method for dialogues, the embodiment support, the design paradigm, the possibility for interaction management and whether the tool supplies design patterns.

Tool	Information Processing	Authoring Interface	Embodiment	Design Paradigm	Dialogue Control	Interaction Management	Design Patterns
DialogFlow, Wit.ai, LUIS.ai, Watson, Lex, Recast.ai	Neural network	Web-interface for intent-entity mapping	No	Pragmatic	Single	No	No
RavenClaw	Hierarchical plan based	Dialogue task specification	Yes	Theory-driven	Single or distributed	Yes	No
Disco	Hierarchical task based	Hierarchical tree authoring	Yes	Theory-driven	Single	Yes	No
VHToolkit	Information state update	NPC Editor FLoReS, scripting	Yes	Pragmatic	Single or distributed	Yes	No
IrisTK	Statecharts	Statecharts in IrisFlow	Yes	Pragmatic	Single	Yes	Some
PyDial	Statistical network	Ontologies, user simulation	Yes	Pragmatic	Single	No	No
OpenDial	Probabilistic information state update	Probabilistic rules	Yes	Pragmatic or theory-driven	Single	Yes	Some
HALEF	Any	VoiceXML	No	Pragmatic	Distributed	Yes	No
TrindiKit	Information state update	Formal declaration	Yes	Theory-driven	Single or distributed	Yes	Some
WAMI	Frame-based	Scripting	Yes	Pragmatic	Single	Yes	No
Visual SceneMaker	Statecharts	Sceneflow and Scenescript	Yes	Pragmatic	Single or distributed	Yes	Some
Flipper 2.0	Information state update	Rules, scripting	Yes	Pragmatic or theory-driven	Single or distributed	Yes	Yes

dialogue flow, maintaining transparency of changes in the dialogue state. In complex dialogues a single component for dialogue control can be a bottleneck. RavenClaw [1], the VHToolkit [10] and Flipper are capable of distributed control, using separate components, for example, for backchanneling and deliberate conversation.

During a conversation with an ECA, turn-taking and backchanneling are important for a coherent conversation; this is called **interaction management**. The commercial tools only support rigid turn-by-turn dialogues; there is no managing of other turn behaviour like pauses or interruptions. Flipper has a structure similar to RavenClaw [1] and IrisTK [21] to support both simple turn-by-turn behaviour as well as more dynamic turn-taking.

Most dialogue design tools provide a description of their tool and simple examples to run the software. However, an underestimated aspect is how precisely to design the dialogues themselves: which **design patterns** a designer of dialogues could use. IrisTK [21] and Visual SceneMaker [7] do provide dialogue flow patterns for authors but design patterns on the higher level dealing with sensory input or behaviours are not provided. These design patterns help authors with fast decision making of prototyping their ECA. In this paper we describe multiple types of design patterns that are helpful in developing dialogues in Flipper.

3 FLIPPER

Flipper is a dialogue engine for pragmatic yet robust dialogue management that is applicable in many domains, and has reusable design patterns. Designers of ECAs can use the dialogue engine to quickly create dialogue systems that can be as complex as they like. The software is open-source and available on GitHub.[4]

The main concepts in Flipper are the information state and declarative templates written in XML. The **information state** can be pre-defined, created at runtime, and/or updated on-the-go. It stores interaction-related information and data in a hierarchical tree-based structure. The information state is represented in JSON format, making it human readable and easy to integrate with other dialogue components that support working with JSON data structures. Listing 1 shows an example information state. In this example the data structure's top-level root node '`is`' has a child node '`agent`' which stores information such as the name of the user, the last recognised user utterance, the current user emotion, and the dialogue history. Nodes in the information state can be accessed in Flipper by navigating the tree-based data structure using dot notation. For example, the user's name can be accessed through '`is.agent.userName`'. Flipper can be linked to a PostgreSQL database to create a persistent information state. This means that the information state can be restored to a previous valid information state that exists in the database. Such a persistent information state can be used, for example, to track interactions with a user over multiple sessions.

The data structure stored in the information state is queried and updated using **templates**. Templates can be grouped and organised in different files according to their related functionality. Each template consists of **preconditions** and **effects**. Preconditions are sets of rules that describe when a template should be executed. Effects are the associated updates to the information state. Listing 2 shows

[4]GitHub link: https://github.com/hmi-utwente/flipper-2.0

an example template that checks whether a user is present. If so, the user is personally greeted. Using the information state from Listing 1, this template will result in the agent saying the following greeting: "Hello, Alan! Nice to meet you!"

Preconditions and effects are evaluated using the Nashorn JavaScript Engine, which supports up to ECMAScript 5.1. In Flipper, JavaScript expressions and functions can be used as an imperative addition to the declarative template approach. Finally, Flipper exposes Java objects to be used within templates for further integration with existing (external) software modules.

```
{ "is" : { "agent" : {
 "userPresent" : true,
 "userName" : "Alan",
 "userSpeech" : "hello what can you do",
 "userEmotion" : "happy",
 "history": {
  "greetByAgent" : false,
  "greetByUser" : false
}}}}
```

Listing 1: An example information state that stores the agent's knowledge of the interaction.

```
<template id="hello_world">
 <preconditions>
  <condition>is.agent.userPresent</condition>
 </preconditions>
 <effects>
  <assign is="is.agent.say">
   "Hello "+is.agent.userName+"! Nice to meet you!"
  </assign>
 </effects>
</template>
```

Listing 2: Example template where the agent greets the user.

3.1 Transaction Model

The dialogue engine uses a **transaction model** to ensure reliability. According to Gray and Reuter [8, p. 6], a transaction is *[...] a collection of operations on the physical and abstract application state.* In Flipper, the check of the preconditions in all templates and execution of their associated effects is considered as one transaction. A transaction is complete when it is successfully committed to a database.

In each transaction, the conditions of all templates are checked on a frozen information state. The effects of the templates that are true are executed consecutively. If all effects are executed successfully, the updated information state is committed to the database. If one of the effects fails, all processed effects in the current transaction are rolled back and the information state is restored to the previous state, which is retrieved from the database. Template checking occurs in recurring intervals. A limit can be set on the frequency with which templates are checked. For example, with a frequency of 20 Hz all templates are checked once every 50 ms. Setting a higher frequency may result in a more responsive system, while setting a lower frequency leads to a lower system load.

4 CREATION OF A DIALOGUE MANAGER

The first important thing to think about when designing dialogues is the information flow of the dialogue. What type of information is needed from the user and when? What type of information is required for the agent? What should the agent do and when? Which behaviours need to be displayed and when? Here we explain how to create a dialogue system with Flipper and showcase some design patterns using the 'sense, think, act' metaphor.

4.1 Sensing

An interactive ECA needs sensory input from the user. This information needs to be put into the information state so that concurrent processes can use it. Flipper itself does not contain sensing components, but projects that include them are available for download (c.f. Section 5).

To receive sensory input from auxiliary devices or software modules we have developed a 'middleware' component. This component is a wrapper around existing off-the-shelf messaging and communication services. Currently Flipper supports wrappers for ActiveMQ, ROS, YARP, Apollo/STOMP, TCP/IP, UDP and USB. Our middleware component listens to messages on a supported communication channel and then places them in the information state. When such messages are received in JSON format they can directly be stored in the information state; otherwise the message has to be preprocessed into a JSON format first.

Once the sensory information has been placed in the information state it has to be processed to determine the impact on the dialogue flow. To prevent templates from processing the same sensor information twice accidentally, we suggest the following design patterns for dealing with sensory input in Flipper.

As a first simple approach, each template could be required to have an effect that negates its own precondition, such as in Listing 3, where the parameter `is.agent.userExpressionEvent` is set from `smile` to `none`. A template could remove the sensor input from the information state once it has processed it. Although this is a pragmatic and quick solution it is not a scalable approach for the long term. Also, it results in verbose templates. When the impact of a new sensor value should be more multi-faceted, one could construct a template file with a collection of templates that first dump the raw input in a temporary information state variable and then successively process the input. Separating the multiple effects of the new sensor input into multiple templates keeps the templates

```
<template id="soc_respond_to_smile" conditional="true">
 <preconditions>
  <condition>
  is.agent.userExpressionEvent === "smile"
  </condition>
 </preconditions>
 <effects>
  <assign is="is.agent.fml.template">"smile_return"</assign>
  <assign is="is.agent.userExpressionEvent">"none"</assign>
 </effects>
</template>
```

Listing 3: Example of a template that removes input once it has been processed.

```
<template id="1">
 <preconditions>
  <condition>is.sensor.userExpressionEvent === "smile"
  </condition>
  <condition>is.sensor.userGesture === "waving"
  </condition>
 </preconditions>
 <effects>
  <assign is="is.agent.fml.template">"smile_and_wave_return"
  </assign>
 </effects>
</template>
<!-- Many more templates could reside here, each triggering
on a combination of is.sensor.userExpressionEvent and
other preconditions-->
<template id="x">
 <preconditions>
  <condition>is.sensor.userExpressionEvent === "smile"
  </condition>
  <condition>is.weatherStation.currentWeather === "sunny"
  </condition>
 </preconditions>
 <effects>
  <assign is="is.agent.speak">"Beautiful day today!"
  </assign>
 </effects>
</template>
<template id="last">
 <preconditions>
  <condition>is.sensor.userExpressionEvent !== "neutral"
  </condition>
 </preconditions>
 <effects>
  <assign is="is.sensor.userExpressionEvent">"neutral"
  </assign>
 </effects>
</template>
```

Listing 4: Quick design pattern for dealing with a sensory input event. The top templates are triggered by a user's detected facial expression and other sensor information. The last template 'cleans up' the sensory input to make sure actions based on such sensory input are only processed once.

relatively clean and readable. The execution order of templates is always defined by the order of templates in the template file. A final template can do a cleanup of the raw sensory input once the other templates have finished. See Listing 4 for an example of such a template file. However, this solution is useful only when developing small behaviours, because with multiple template files it is impossible to be sure which template is executed last.

Another design pattern for dealing with sensory input is to keep track of a history of sensory input and check against time or sensor value index whether the input has been processed already. This can be done by either keeping track of an index or a timestamp. The downside of this approach is that it creates more overhead (more memory consumption) and is more complex to implement than the other two pragmatic approaches. However, for robust and scalable systems where you cannot be sure which other template sets might have access to the same information, this last approach

```
<template id="add new">
 <preconditions>
  <condition>isNew(is.sensor.userEmotion)</condition>
 </preconditions>
 <effects>
  <assign is="is.agent.history.emotions">
   addToArray(is.agent.history.emotions,
              is.sensor.userEmotion)
  </assign>
 </effects>
</template>

<template id="remove old">
 <preconditions>
  <condition>isFull(is.sensor.userEmotion)</condition>
 </preconditions>
 <effects>
  <assign is="is.agent.history.emotions">
   removeHeadArray(is.agent.history.emotions)
  </assign>
 </effects>
</template>
```

Listing 5: Complex design pattern for dealing with input.

is a necessity. Listing 5 shows an example of templates dealing with sensory input in this way. We emphasize that each of these solutions can be applicable in a specific case and that each pattern is a good approach for working with Flipper depending, among other things, on which stage of development the dialogue system is in.

```
<template id="soc_sal_returnsalutation" conditional="true">
 <preconditions>
  <condition>!is.agent.greetByAgent</condition>
  <condition>is.agent.userEmotion === "happy"</condition>
  <condition>containsKeyword(is.agent.userSpeech,
                             ["hello","hi"])</condition>
 </preconditions>
 <effects>
  <assign is="is.agent.fml.template">
  "social_salutation_return"</assign>
  <assign is="is.agent.fml.parameters['var.name']">
  is.agent.userName</assign>
  <assign is="is.agent.fml.parameters['emotion.em1']">
  is.agent.userEmotion</assign>
  <assign is="is.agent.greetByAgent">true</assign>
 </effects>
</template>
```

Listing 6: Example of a set of templates returning a user's happy greeting.

4.2 Thinking

Information from the input can be used by the agent to 'think' about it in order to determine an appropriate response in the current dialogue context. This is done in what we call 'dialogue behaviour templates'. An example of a dialogue behaviour template is shown in Listing 6. This template covers the situation when a user has not

```
<template id="behaviour">
 <preconditions>
  <condition>is.agent.behaviours.length !== 0</condition>
 </preconditions>
 <effects>
  <behaviour name="executeBehaviour">
   <object class="behaviourRealiser" persistent="behaviourRealiser"></object>
   <arguments>
    <value class="String" constant="<bml id='bml1' xmlns='http://www.bml-initiative.org/bml/bml-1.0'
    character='Alice'><gaze id='gaze1' target='PERSON1'/></bml>"/>
   </arguments>
  </behaviour>
 </effects>
</template>
```

Listing 7: Template sending a String message in BML format with gaze behaviour to a behaviour realiser.

previously been greeted by the agent, the user is currently happy, and the user has said 'hello' or 'hi'. In this case the agent should return the greeting with a friendly face, including the user's name.

We encourage designers to make a distinction between high-level and low-level interaction templates. This recommendation follows Lemon et al. [13], who describe their dialogue design approach as creating high-level dialogue (content) moves, but also handling low-level (management) phenomena like turn-taking, back channelling, and grounding. Turn-taking for example can be done by a state-machine which regulates turns based on current speech activity of the user and agent. By using this conceptual division between content and management templates —a design distinction only; Flipper does not register a formal distinction between the two— some management templates can be reused in different ECAs and different projects. For example, components that contain low-level information state updates for turn-taking and back channelling are applicable in multiple domains and can be used in each agent that requires it, whereas high-level content templates are often not reusable as they contain domain-specific content.

4.3 Acting

We have also developed modules for Flipper that use our middleware component to communicate with the behaviour realiser of an embodied conversational agent. These modules can send both Behavioural Markup Language (BML) and Functional Markup Language (FML) [27]. Inside Flipper, the parameters for the behaviours need to be determined and set accordingly in a valid BML or FML representation, depending on the behaviour realiser.

A pragmatic way to deal with agent behaviours is to specify BML or FML strings directly inside templates. Listing 7 shows an example of this. The `behaviourRealiser` is a Java module specifically designed for sending BML and FML behaviours of the agent via our middleware component to an external behaviour realiser.

An alternative approach is to create a list of BML or FML behaviours. These behaviours can be loaded in the dialogue system from the file system. Authors can use existing BML behaviours accompanying the Flipper software or create their own. Additionally, these behaviours can be parametrised and the parameters can be filled using the information from the information state (see Cafaro et al. [3]). In Listing 6 the assignments of `is.agent.fml.parameters` include setting the name of the user interacting with the system and the emotion of the user. Once the parameters are set, the behaviour can be sent through our middleware component to a (BML or FML compliant) behaviour realiser, as shown in Listing 8.

4.4 Advanced Dialogue Behaviour

Authors might require extra functionality in the design of their dialogues. As the dialogue engine evaluates the templates with JavaScript, it is easy to add existing JavaScript libraries or JavaScript code to perform logic that is cumbersome to express in (declarative) templates. One example applicable to an embodied conversational agent is a function that checks certain keywords in an utterance (shown in Listing 9). Other useful JavaScript functionalities are behavioural generators, calculating the appropriate intensity of an emotion of an agent, and timers necessary to know when to perform certain behaviours.

When JavaScript is not expressive enough or when the JavaScript becomes too large to author or maintain, Java classes can be instantiated and integrated in the dialogue engine. Complex functions can be delegated to Java objects that have been created from within the Flipper template collection. This further extends the capabilities and flexibility of Flipper. Examples of useful Java modules are the StanfordCoreNLP for natural language understanding [16] and BML translators such as ASAP for behaviour generation [26].

Connecting with non-Java external components can be done by sending messages across a middleware channel, as mentioned earlier. Information can be exchanged between such external components and dialogues, and retrieved from or stored in the information state. For instance, external reasoners, knowledge bases or natural language generators can interface with Flipper via the supplied middleware and use a separate template file for the handling of their input and output to keep the system modular and reusable. Additionally, this connectivity includes external components that are not 'traditional' embodiments for conversational agents, for example a tablet that displays information or an external device (e.g. a coffee machine) that is started automatically when the user requests this in the dialogue with the ECA.

```
<template id="executeBehaviour">
 <preconditions>
  <condition>is.agent.fml !== ""</condition>
 </preconditions>
 <effects>
  <behaviour name="executeBehaviour">
   <object class="behaviourRealiser" persistent="behaviourRealiser"></object>
   <arguments>
    <value class="String" is="is.agent.fml" is_type="JSONString"/>
   </arguments>
  </behaviour>
  <assign is="is.agent.fml">""</assign>
 </effects>
</template>
```

Listing 8: Template that takes an FML request plus its parameters and passes it to the `behaviourRealiser` module for execution.

```
function containsKeyword(utterance, keywords){
 var word;
 list = utterance.split(" ");
 for(word in list){
  var key;
  for(key in keywords){
   if(keywords[key] === list[word]){
    return true;
 }}}
 return false;
}

function addToArray(array, value){
 array.push(value);
 return array;
}

function removeHeadArray(array){
 array.shift();
 return array;
}
```

Listing 9: Example JavaScript functions, one that checks for keywords in a user utterance, and two for addition and removal operations in arrays.

5 USE-CASES

In this section we discuss some of the projects that have used Flipper in developing their agent(s). The projects use different types of input and output modalities that are connected to the dialogue engine.

In the ARIA-VALUSPA project, a multi-modal information providing agent was developed [25]. Users can ask the agent questions about a specific domain and the agent tells stories to the users. The agent also includes an emotional model that determines whether the agent likes or dislikes the user, based on the user's (non-) verbal responses. Turn-management, behaviour generation and the emotional model are handled by Flipper templates, whereas external components are used for natural language understanding and (non-) verbal behaviour realisation.

In another project involving multi-agent parties, an external module for synchronisation of behaviours was developed and integrated with Flipper templates to create social gaze behaviour based on saliency [11]. Saliency indicates what is important during interactions; for example, most gazes will be directed towards the speaker in the current interaction [20].

The Snoozle project aimed at helping people sleep using an interactive pillow that lured people to bed. Flipper was used to steer the multi-modal behaviour of the pillow [28]. This is an example where a non-humanoid agent is controlled by Flipper.

Some of the proposed design patterns result from lessons learned in projects using the first version of Flipper [22]. In the R3D3 project, Flipper was used for steering the turn and emotive behaviour of a receptionist robot combined with a virtual agent [23]. In the DE-ENIGMA project, involved with child-robot interaction, Flipper was used to control the behaviour (speech, facial expressions and gestures) of an emotionally expressive robot. Additionally, modules were developed for a dialogue logger and a dialogue tree within Flipper [5]. Finally, in another child-robot interaction project called EASEL, Flipper was used to control actuated physical learning materials as well as a tablet displaying the GUI of an educational game [18].

6 FUTURE WORK

Flipper 2.0 is under active development in the context of several European research projects. We are working on creating a debugging functionality that will give insight into exactly what state the dialogue is in and when certain information state updates will be applied. Additionally, we want to supply Flipper 2.0 with basic modules for an operational ECA to have a prototype system ready out of the box. The DE-ENIGMA project will be updated to use Flipper 2.0 and another project called Council of Coaches[5] already uses Flipper 2.0 in its prototype. The Council of Coaches project focuses on coaching dialogues using a virtual multi-agent platform.

Flipper currently has no graphical user interface for editing the required templates. An editor for modifying template files is on its way, to prevent users from making syntax errors.

[5] http://council-of-coaches.eu/

One might think that the rule-based approach used in Flipper is too simple for designing a dialogue system. However, we see machine learning possibilities for the dialogue engine as well. Speech and text oriented approaches using machine learning often require the collection of conversations or the authoring of input/output behaviour. Similarly for Flipper, an author could collect information state mappings between sensory input and user behaviours that map to certain agent behaviours, or author information states and behaviours to let a computer learn the most appropriate agent behaviour over time. Another option is to connect external machine learning models for specific low-level management tasks such as turn-taking and use this information in high-level templates.

Scalability might be a problem if many update rules need to be integrated. However, we view the reusability of modules as one particular case of scalability for designing dialogues. In the case where an author designs a dialogue system that needs large amounts of data and is open-domain, we suggest to use Flipper for low-level interaction dialogue strategies in combination with, for instance, a cloud-based commercial tool or PyDial.

7 CONCLUSION

We have provided some insight into the development process of dialogues for embodied conversational agents (ECAs) in complex projects, and have presented Flipper 2.0: a tool that makes it easy to quickly and iteratively create dialogues for ECAs, meeting the demands of such projects. This tool is particularly useful for people creating dialogues who need to get started quickly, with workable and pragmatic dialogue patterns, yet need to have the possibility to extend their efforts into a complex, multi-faceted, responsive, multi-modal dialogue system.

8 ACKNOWLEDGEMENTS

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreements No. 645378 and No. 688835, and the European Union 7th Framework Program (FP7-ICT-2013-10) EASEL under the grant agreement No. 611971.

REFERENCES

[1] Dan Bohus and Alexander I Rudnicky. 2003. RavenClaw: Dialog management using hierarchical task decomposition and an expectation agenda. In *Eighth European Conference on Speech Communication and Technology*.

[2] Daniel Braun, Adrian Hernandez-Mendez, Florian Matthes, and Manfred Langen. 2017. Evaluating natural language understanding services for conversational question answering systems. In *18th Annual SIGdial Meeting on Discourse and Dialogue*. 174–185.

[3] Angelo Cafaro, Merijn Bruijnes, Jelte van Waterschoot, Catherine Pelachaud, Mariët Theune, and Dirk Heylen. 2017. Selecting and Expressing Communicative Functions in a SAIBA-Compliant Agent Framework. In *Intelligent Virtual Agents*. 73–82.

[4] Massimo Canonico and Luigi De Russis. 2018. A comparison and critique of natural language understanding tools. In *Cloud Computing 2018*. 110–115.

[5] Pauline Chevalier, Jamy Jue Li, Eloise Ainger, Alyssa M. Alcorn, Snezana Babovic, Vicky Charisi, Suncica Petrovic, Bob Rinse Schadenberg, Elizabeth Pellicano, and Vanessa Evers. 2017. Dialogue design for a robot-based face-mirroring game to engage autistic children with emotional expressions. In *International Conference on Social Robotics*. 546–555.

[6] Adam Cheyer and David Martin. 2001. The Open Agent Architecture. *Autonomous Agents and Multi-Agent Systems* 4, 1-2 (March 2001), 143–148. https://doi.org/10.1023/A:1010091302035

[7] Patrick Gebhard, Gregor Mehlmann, and Michael Kipp. 2012. Visual SceneMaker — a tool for authoring interactive virtual characters. *Journal on Multimodal User Interfaces* 6, 1-2 (2012), 3–11.

[8] Jim Gray and Andreas Reuter. 1992. *Transaction processing: concepts and techniques*. Elsevier.

[9] Alexander Gruenstein, Ian McGraw, and Ibrahim Badr. 2008. The WAMI toolkit for developing, deploying, and evaluating web-accessible multimodal interfaces. In *10th International Conference on Multimodal Interfaces*. 141–148.

[10] Arno Hartholt, David Traum, Stacy C. Marsella, Ari Shapiro, Giota Stratou, Anton Leuski, Louis-Philippe Morency, and Jonathan Gratch. 2013. All together now: Introducing the Virtual Human Toolkit. In *Intelligent Virtual Agents*. 368–381.

[11] Jan Kolkmeier, Merijn Bruijnes, Dennis Reidsma, and Dirk Heylen. 2017. An ASAP Realizer-Unity3D bridge for virtual and mixed reality applications. In *Intelligent Virtual Agents*. 227–230.

[12] Staffan Larsson and David Traum. 2000. Information state and dialogue management in the TRINDI Dialogue Move Engine Toolkit. *Natural Language Engineering* 1, 1 (2000), 1–17.

[13] Oliver Lemon, Lawrence Cavedon, and Barbara Kelly. 2003. Managing dialogue interaction: A multi-layered approach. In *4th Annual SIGdial Meeting on Discourse and Dialogue*.

[14] Anton Leuski and David Traum. 2011. NPCEditor: A tool for building question-answering characters. In *International Conference on Language Resources and Evaluation (LREC)*. 2463–2470.

[15] Pierre Lison. 2015. A hybrid approach to dialogue management based on probabilistic rules. *Computer Speech and Language* 34, 1 (2015), 232–255.

[16] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. 2014. The Stanford CoreNLP natural language processing toolkit. In *ACL 2014, System Demonstrations*. 55–60.

[17] Vikram Ramanarayanan, David Suendermann-Oeft, Alexei V Ivanov, and Keelan Evanini. 2015. A distributed cloud-based dialog system for conversational application development. In *16th Annual SIGdial Meeting on Discourse and Dialogue*. 432–434.

[18] Dennis Reidsma, Vicky Charisi, Daniel Davison, Frances Wijnen, Jan van der Meij, Vanessa Evers, David Cameron, Samuel Fernando, Roger Moore, Tony Prescott, Daniele Mazzei, Michael Pieroni, Lorenzo Cominelli, Roberto Garofalo, Danilo De Rossi, Vasiliki Vouloutsi, Riccardo Zucca, Klaudia Grechuta, Maria Blancas, and Paul Verschure. 2016. The EASEL project: Towards educational human-robot symbiotic interaction. In *Living Machines 2016: Biomimetic and Biohybrid Systems*. 297–306.

[19] Charles Rich and Candace L. Sidner. [n. d.]. Using collaborative discourse Theory to partially Automate dialogue tree authoring. In *Intelligent Virtual Agents*. 327–340.

[20] Kerstin Ruhland, Christopher E Peters, Sean Andrist, Jeremy B Badler, Norman I Badler, Michael Gleicher, Bilge Mutlu, and Rachel McDonnell. 2015. A review of eye gaze in virtual agents, social robotics and HCI: Behaviour generation, user interaction and perception. In *Computer Graphics Forum*, Vol. 34. 299–326.

[21] Gabriel Skantze and Samer Al Moubayed. 2012. IrisTK: a statechart-based toolkit for multi-party face-to-face interaction. In *14th International Conference on Multimodal Interaction*. 69–76.

[22] Mark Ter Maat and Dirk Heylen. 2011. Flipper: An information state component for spoken dialogue systems. In *Intelligent Virtual Agents*. 470–472.

[23] Mariët Theune, Daan Wiltenburg, Max Bode, and Jeroen Linssen. 2017. R3D3 in the Wild: Using A Robot for Turn Management in Multi-Party Interaction with a Virtual Human. In *IVA Workshop on Interaction with Agents and Robots: Different Embodiments, Common Challenges*.

[24] Stefan Ultes, Lina M. Rojas Barahona, Pei-Hao Su, David Vandyke, Dongho Kim, Iñigo Casanueva, Paweł Budzianowski, Nikola Mrkšić, Tsung-Hsien Wen, Milica Gasic, and Steve Young. 2017. PyDial: A multi-domain statistical dialogue system toolkit. In *ACL 2017, System Demonstrations*. 73–78.

[25] Michel Valstar, Tobias Baur, Angelo Cafaro, Alexandru Ghitulescu, Blaise Potard, Johannes Wagner, Elisabeth André, Laurent Durieu, Matthew Aylett, Soumia Dermouche, et al. 2016. Ask Alice: an artificial retrieval of information agent. In *18th International Conference on Multimodal Interaction*. 419–420.

[26] Herwin van Welbergen, Ramin Yaghoubzadeh, and Stefan Kopp. 2014. AsapRealizer 2.0: The next steps in fluent behavior realization for ECAs. In *Intelligent Virtual Agents*. 449–462.

[27] Hannes Vilhjálmsson, Nathan Cantelmo, Justine Cassell, Nicolas E Chafai, Michael Kipp, Stefan Kopp, Maurizio Mancini, Stacy Marsella, Andrew N Marshall, Catherine Pelachaud, et al. 2007. The behavior markup language: Recent developments and challenges. In *Intelligent Virtual Agents*. 99–111.

[28] Jered Vroon, Cristina Zaga, Daniel Davison, Jan Kolkmeier, and Jeroen Linssen. 2017. Snoozle–A robotic pillow that helps you go to sleep: HRI 2017 Student Design Competition. In *Companion of the 2017 ACM/IEEE International Conference on Human-Robot Interaction*. 399–400.

View publication stats